14792

Prognostics Models of Combat Vehicles Software

Elena Bankowski and Abul Masrur

US Army RDECOM-TARDEC, Warren, MI 48397-5000

ABSTRACT

The Next Generation Software and Survivability Technology areas of TARDEC RDECOM proposed the Dependable Automated Reconfigurable Technology (DART). The DART's "Health & Situation Control" will test the processing elements with Probe/Agent technology for software checking. Algorithms within the Health & Situation Control will assess the health of the processors and recommend element hand-off based on a "Criticality Scoring System" in conjunction with the Statistical Usage Test (SUT) model. The DART technology represents the next generation of software systems for ground combat vehicles. DART will enhance the performance of a weapon system by providing on-the-fly reconfiguration to accommodate the loss or malfunction of processing elements or to optimize onboard computational capability. Off-vehicle probes will be launched to assess the health of companion vehicles within the Operations Unit. The SUT will be used to evaluate software reliability. The SUT combined with a test environment that includes test benches, simulators and automated testing will provide the ability to arrive at a statistically valid measure of the reliability of the software. The SUT methodology will enhance software development and test processes. The end result will be the increased reliability of fielded software intensive systems.

INTRODUCTION

Modern embedded systems are getting more and more software intensive, and successful operation of a system requires high fidelity not only in its hardware components, but also in the software used in such systems. Successful prognostics in software is a challenge to the technical community and the technology in is in its infancy. Researchers have reported some work on reliability of software systems [1-7]. These imply that given a software module, some test will be conducted on those, and based on the results, a measure of reliability will be calculated. Reliability can be considered to be the probability that a software will function the way it was meant to, under a given operational profile (input/output relationship) during a certain time frame. Reliability can be considered to have a one to one correspondence with prognostics. Let's say a software is meant to be operational from its inception till the system is disposed of, which is a duration of 10000 clock hrs. At the end of 2000 clock hrs, let's say by some means we found that its reliability is only 90%. That will imply that for the remaining 8000 clock hrs, it most likely will be available for actual operation for only 8000 * 0.9 = 7200 hrs. So, in a conservative scenario, the system should be replaced after 9200 hrs from the initial commission of the system. Here we have translated the reliability index of 0.9 into a prognostics index of 7200 hrs. for the remaining period of life cycle, or 9200 hrs from the initial commission. Hence prognostics and reliability are very much related.

Software in an embedded system, as in a vehicle, which can be a battle tank as in our discussion later, resides in various processors within the vehicle. Therefore, to find the health status of the embedded system, it is necessary to find the health situation of the processors (hardware) and the software which resides in it. Having found the health situation, it can, in certain situations, be possible to reconfigure an embedded system, by reconfiguring its hardware and software architecture. This can lead to a more gracefully degradable mode for the system, before the situation permits a thorough checkup and off-line replacement in a service facility, thus leading to higher survivability of the system. This paper will mainly focus on software prognostics and reliability, leading to reconfiguration by DART method, based on SUT based testing. Information about the system will be collected by various hardware sensors and software probes (request for various data using software algorithms), as detailed later.

Prior art - As noted earlier, software reliability has been discussed by certain researchers. In order to find a measure of reliability, some modeling techniques for reliability have been proposed. Horgan et. al [1] has suggested that establishing the operational profiles i.e. input domain of the software (possible input states and their probability of occurrence) and customer input related to usage, can be a formidable task. Methodology for this have been provided by Musa [2]. It has been indicated [1] that if the test method is weak i.e. fails to identify all the features and path in the software properly, then it can be inadequate. Obviously such inadequacy can lead to optimistic and erroneous results in the reliability estimation and hence prognosis evaluation. Nelson [3] has suggested a statistical testing which calls for randomly generated test cases using a statistical distribution based on operational

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **04 APR 2005**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **Prognostics Models of Combat vehicles Software**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **Elena Bankowski; Abul Masrur**		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **USA TACOM 6501 E 11 Mile Road Warren, MI 48397-5000**		8. PERFORMING ORGANIZATION REPORT NUMBER **14792**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) **TACOM TARDEC** 11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release, distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**SAR**	**7**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

profile. It seems that none of the methodologies are perfect and that a combination of methods involving such statistical testing along with code analysis [4] may provide better results in software reliability estimation. It has been noted [1] that these methods are not used in practice in spite of their merits. Our work, therefore, shows applications where these could potentially be used. In addition, for embedded systems, it is not sufficient to just estimate reliability and come up with a prognosis. If there is a safety critical implication, appropriate methodologies should be adopted to address the situation through reconfiguration wherever possible. In our work, which relates to the reliability and prognostics of embedded hardware and software based military vehicular systems, we therefore present in the following the methodologies for testing (SUT) embedded software, and then the mechanism for following up the findings of the testing through dynamic reconfiguration (DART) when needed.

MILITARY VEHICULAR APPLICATIONS

Today's Main Battle Tanks (MBT) contain a multitude of processors, yet systems such as the Abrams tank provide redundancy only between the hull electronics unit and the turret electronics unit. The Abrams employs duplicate processors hosting redundant software in different vehicle compartments. Over a half a million lines of software code span multiple processors. Future weapon platforms will host a significant increase in software. The processing burden of the front line vehicles will require a further increase in processing capability. Next generation weapon systems processing requirements will grow with the incorporation of intelligent decision aids, sensor fusion, and advanced communications. A future system will have much more configuration combinations than today's MBT. Cost, reliability, space, and mission requirements will preclude achieving redundancy with dedicated, embedded processors that duplicate functionality. The next generation systems vision is that a collection of general-purpose processors connected to a common bus will be scattered throughout the vehicle and assigned dynamically to the various vehicle control and mission-specific tasks as required. This approach, shown in Figure 1, reduces cost and provides greater effective redundancy, since any healthy processor can be assigned to any task instead of having a limited number of backups of ever decreasing capability. Line Replaceable Units (LRU's) are: engine, displays, electronic control units, etc.

Next generation systems process requires extensive monitoring and analysis capabilities to track whether the weapon system is operating properly. A robust reconfiguration capability is required to gracefully and rapidly reorganize the assignment of tasks to processors to respond to hardware and software failures and to changed mission requirements (e.g., switching from surveillance mode to combat/engagement mode). The software in the Abrams tank is created in a highly sophisticated development and testing environment. The U.S. Army Next Generation Software Engineering Technology Area began implementing Statistical Usage Testing (SUT) [5] in the Post Production Software Support (PPSS) project for the U.S. Army's main battle tank. The pilot project on the Abrams M1A2 had the following objectives:

- the determination of how SUT can improve M1A2 field reliability;
- the development of usage model(s) for the Driver's Integrated Display (DID);
- development and documentation of a tailored modeling process to allow for scale-up of SUT, including guidance on modeling practices; investigation of tie-in of SUT with existing PPSS testing and future research.

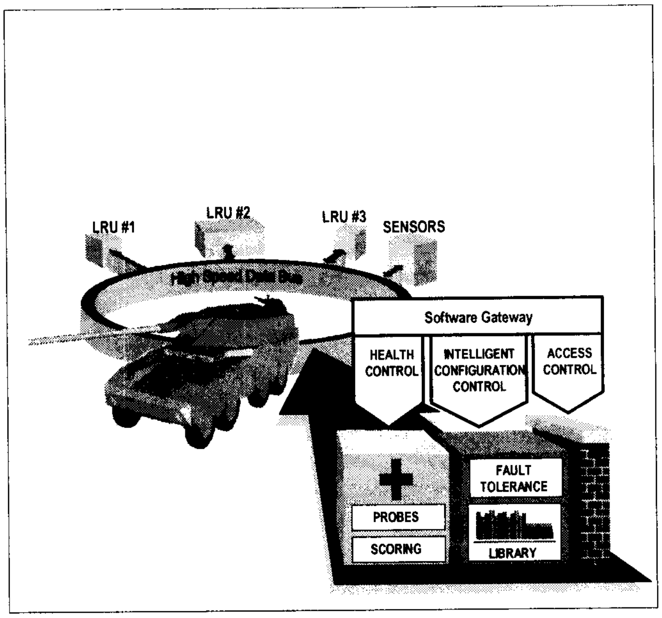

Figure 1: The Next Generation Dynamic Software Assembly.

Usage model development for the DID was facilitated by the toolset Certify® which supported Markov chain usage model development, test management, test case generation and statistical testing. A prototype tool for composing top-level models and lower-level sub-models into one flattened toolset Certify® model was also used. This Model Compose utility allowed for the development of sub-models similar to subroutine development in programming. The major lessons learned from the SUT project were:

- SUT can positively impact PPSS testing, since the focus is on operational usage;
- usage modeling is feasible in the M1A2 PPSS environment;
- usage modeling uncovers a number of issues that relate to behavior and testing;

- A logical and complementary relationship exists between the current testing approach used by the Next Generation Software Technology Area of TARDEC and the SUT methodology.

The method and results of a pilot project were discussed in a paper [5]. The SUT modeling techniques were applied to the Driver's Integrated Display (DID), a component of the soldier-machine interface of the tank. Since then, additional LRU's with increasing complexity have been modeled using SUT. In applying modeling techniques, a high degree of complexity was observed, consisting of the numbers of screens to be modeled and the amount of information that could impact a tester's next action. These challenges were overcome using some innovative approaches.

The Next Generation Software Technology Area of TARDEC investigated the feasibility of using SUT in the PPSS test environment. The primary motivation was the realization that there were not enough test assets, people or time to test the main battle tank software for each release. US Army TACOM personnel completed training in SUT. Q-Labs (army contractor) had expertise and provided mentoring. Next Generation test staff and Q-labs developed practical techniques to link current black box, and regression test procedures with SUT tools and models. Software Engineering Technology (SET, acquired by Q-Labs) delivered SUT training. The purpose of the experiment was to determine better ways to test increasingly complex systems. The paper [5] reported the work performed in 1998-1999. Significant progress has been made since that time. Efforts are now well underway to combine SUT with a number of other approaches and scale up the approach even further.

THE SUT METHOD

The SUT modeling process was proposed by the Q-Labs. The steps of the SUT modeling process were tailored for the Next Generation situation and for easier hand-off in the future. These steps are shown below:

1. Define testing goals – The goals to be defined include the boundaries of what will be tested, reliability goals, etc. These goals appeared in the Test Plan.
2. Define a 'use' – The use was defined as the start and end of a test case. It related to some 'real-world' interpretation of usage, as the testing results are to be used to make inferences on future usage. These tests ensured that the functionality was indeed independent. This also appeared in the Test Plan.
3. Build List of Stimuli – Build the list of all things that enter through the boundary defined in the previous step. The organization of the stimuli list was critical. Grouping the inputs by screen, protocol, or other factor enhanced readability and usability. 'Exceptional' stimuli included time lags, for the situation where certain actions did not occur until a defined time period had passed. They were also used in the models as needed, and appeared in the stimuli list.
4. Define usage variables – All factors that modify the possibility/probability of inputs are listed here. Possible values for each factor are also enumerated. Typically this list was larger than the final list of usage variables, since some usage variables were abstracted away.
5. Define models to develop based upon usage variables and testing goals – The number of 'unique' models was defined here. Significant factors include the potential size of models, potential yield in testing variants in certain areas, etc.
6. Decompose model(s) into sub-models – Defined a rule/approach by which sub-models were created. This was based upon screens, functionality, model distribution or other factors. While doing this, the rules by which lower level models were composed with higher level models were defined. The key was that all the models/sub-models were built consistently so that testing could go on with a single, smooth approach.
7. Build 'single variable' state transition diagrams – For each usage variable state transition diagrams were built. These diagrams defined the manner in which that usage variable could have changed the state. It only included the stimuli that could have caused state changes for the usage variable.
8. Build sub-models and models – The Models were built one at a time, using the Microsoft Word template. All modifications in the models, except for the assignment of probabilities, were done in Word. Models were created by working from the bottom up. Models had four types of states: Entry/Exit, Model States, Composition States, and Collector States. There was one Entry and one Exit state for each model. Model states were typical states of a usage model. Each model state represented a different case of use and a different input of a possibility/probability. Composition states were those which were replaced by a sub-model. Collector states addressed the situations where multiple identical arcs were identical exit arcs of the model. The Collector state had a single exit arc, making model composition less complex. Models addressed all usage variables that could have been impacted within the model or its sub-models.
9. Review sub-models and models – Each model was reviewed to ensure that the model was constructed correctly, appropriate state transitions were made, correct entry and exit states were defined, and lower level models were properly defined.
10. Do initial model composition – We had to perform conversion and transformation first to convert models from Word files into tool SET Certify. Check/analysis was performed next to ensure that models were structurally valid. Finally, we checked that models were composed. This was done to ensure that Composition states had the right

numbers of entry and exit arcs same as sub-model actually had.

11. Determine expected outputs – When tests were executed, the tester had to ensure that two things occurred (in ascending order). First, that the correct state transition in the usage model was made. Second, that the correct response was issued within the performance requirements defined for the system. The correct response and performance for each state transition was documented.
12. Insert probabilities into models/sub-models – Insertion of probabilities that were consistent with the definition of use and test goals. Two or three top arcs were selected at each state, and the probabilities were assigned to these arcs. The rest of the weight was uniformly distributed across the remaining arcs. Probability information was gathered; this information was inserted directly into the tool SET Certify models.
13. Compose models and sub-models – Once probabilities have been defined, a final composition could occur for each set of models. The model composition utility was implemented to achieve this objective. Models were composed from the bottom up. All intermediate compositions were available, along with the fully composed models. Figure 2 shows an example of the SUT states transition diagram – a top-level model.

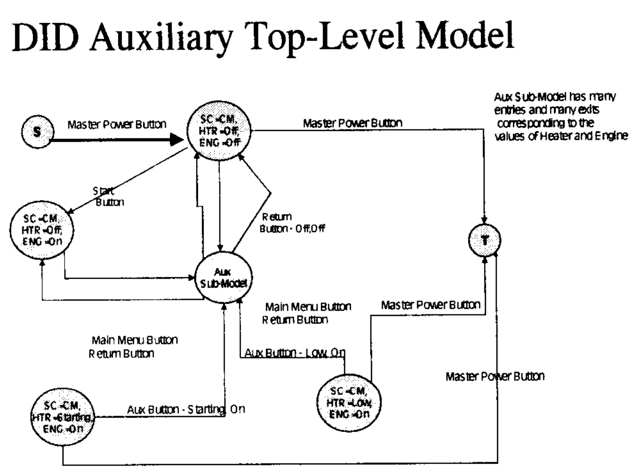

Figure 2: The SUT state transition diagram - an example of a Top-Level model.

14. Conduct test – Tests were executed and the results were compared to actual results.
15. Compile test results – Passed tests, failed tests, and unresolved/un-executed test were recorded in tool SET *Certify*.
16. Analyze test results – Quality and stoppage criteria information were computed by tool SET *Certify*, once the test results have been entered.
17. Make decisions based upon test results.

We adapted the SUT methods for health monitoring and diagnostics of selected line replaceable units of ground combat vehicles, such as driver's integrated display (DID).

THE DART METHOD

The DART's health and situation control continually tests the processing elements with Probe/Agent technology. Algorithms within the Health & Situation Control assess the health of the processors based on a criticality scoring system that considers mission requirements. Software probes launched by the DART controller query processing elements.

Probes are able to capture events that occur in the software systems. They must be able to be inserted automatically to avoid software rework simply to install the probes. A variety of probes will be necessary in order to capture the various kinds of information desired. For example, probes that inspect a software environment checking for the presence or absence of required software are likely to be very different that probes that capture functions calls or messaging information. The probes must have negligible impact on the behavior of the software; particularly they must not interfere with operations by slowing response to a noticeable extent. While this is an issue in conventional testing where the act of monitoring changes the timing behavior, in testing environment it is misleading, whereas in operational environments it could be catastrophic. The solution to this is three fold:

- Probes must be engineered to have minimal space and performance impact.
- Performance thresholds must be defined and monitored for critical software interactions.
- It must be possible to turn off monitoring when these thresholds are in danger of being violated.

Unlike testing environments, in operational environments the testing model is only implied. Thus, in addition to capturing the software interactions, probes must capture external stimuli from the environment and the users. In a test environment, these would have been generated by the test harness and recorded directly from here.

Gauges analyze probed event streams and report summaries/conclusions in a more usable fashion. Gauges output can be used in four distinct ways:

- Gauges can feed back to previous software activities. Example of the feedback included detailed graphs of the developed software configuration compared to design specification, violations of timing constraints, logs of software exceptions thrown, summaries of functions calls, CPU usage of various modules, etc.
- Gauges can be used to affect previous software lifecycle steps. A prime example is the use of gauges to validate that a testing model conforms to reality as observed by probes of the environment. Specifically, if testing is conducted according to a Marko model defining environmental patterns of

stimuli and user responses, it would be desirable to validate that observed patterns are statistically compatible with such a model.

- Gauges can control the operation of the target software, possible by triggering a reallocation of resources, shutting down nonessential functionality, and even compensating for errors.
- Gauges can control the data collection activities by activating and deactivating probes to avoid impacting performance or to focus collection activities on suspect or critical areas.

The probed data is sent to a gauge that has a variable sensitivity or gain. Statistical Usage models and criticality scoring control the sensitivity of the gauge. In response to the gauge, the replicating process launches agents that can insert anomalous events for diagnostic purposes. In this context, a probe is a subset of an agent having only the ability to query without affecting framework, I/O protocol or Quality of Service. Each weapon system fitted with a DARTS Controller will control self-repair and reconfiguration of on-board processors utilizing a statistical based intelligent scoring system. It considers criticality of the function in the current battlefield situation.

A Modern Weapon System is composed of many processing elements as shown in Figure 3. The processing elements are grouped into seven categories, shown as pie segments. Typically, each processing element represents a software program that operates on a dedicated processor. Most often, the elements are co-located inside of the LRU that contains three to nine processors. Current vehicles have more than half of the total processing activity performed on embedded processors.

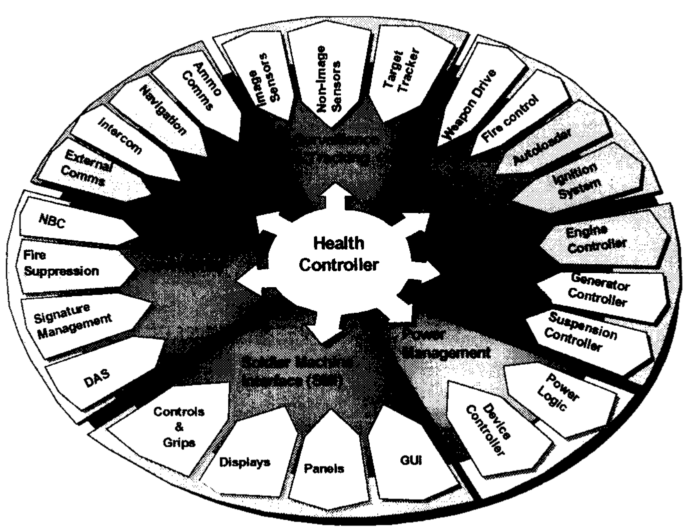

Figure 3. Embedded Processing System's Health Controller.

The Health and Situation Controller, see Figure 4, provides the Probe/Agent Controller, Health and Criticality Scoring and the Auditor Function. The Health and Situation Controller provides the following capabilities that contribute to system assurance.

- **Probe/Agent Controller:** This activity will determine the frequency, format, type and distribution of queries that the Health Controller will send to the elements so that element health can be determined.
- **Health Scoring:** This activity will evaluate collected and processed data to determine if it lies within the output bands of the element.
- **Criticality Scoring:** Based on the nature of the mission, and the battlefield situation, this activity will prioritize vehicle functions that require processing.
- **Auditor:** This activity will compile and report the collective operational and health status of all of the elements to the Health Controller.

This health monitoring experiment provides software parallels to hardware survivability. The software attacks were encountered in a cyber battlefield. Health monitoring provides protection and compartmentalization of software with the same diligence applied to armor protection of hardware.

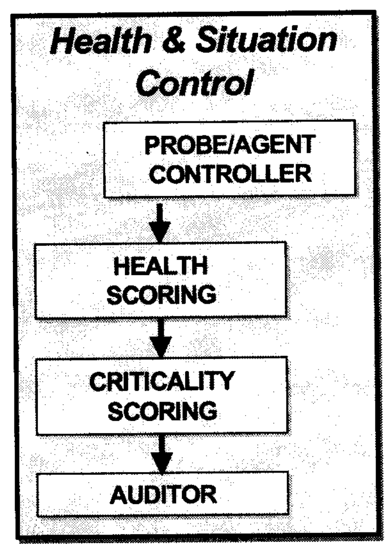

Figure 4. Health and Situation Controller.

The following conditions will require the Health Controller to reconfigure the system:

- Software Failure
- Hardware Failure (Catastrophic such as Battle Damage)
- Hardware/Software Failure (Graceful Degradation)
- Hardware Installation
- Hardware Removal
- Software Installation or Upgrade
- Change in Mission (In-route System Optimization)
- Reorganization of Operations Unit: Change in complexion of Unit. Example:
- Re-assignment of Weapon System Duty Role
- Transfer of TOC
- Training and Simulation.

The reconfiguration process for the first condition "Software Failure" is shown below. Reconfiguration due to "Software Failure" has the following steps:

tep 1: A probe detects a symptom of a software-manifested failure. The Health Controller determines that probe reconnaissance has returned data that is out of the operational bands of the processing element. Figure 14 illustrates a flow chart of the Health Check Process.

Step 2: The Health Controller algorithmically computes the next viable state of the software architecture considering the processing assets available and mission requirements.

Step 3: The Health Controller dispatches a new operational publication of the failed software to a new host processor. Once the transfer is complete, the surrogate is brought on line with a replicating processing effecting repair/replacement of the incumbent. The Content Repository issues the operational publication. Contained in the Content Repository are programs or program objects that dedicated processors host (For example: Ballistic Program, Auto Target Tracker, and Engine Controller).

Step 4: Persistent object data is either extracted from the defective element or retrieved from a synchronized replicator and transferred to the new host (surrogate) as part of the replicating process. If the defective element fails catastrophically, object data will have to be reinstated with default values or acquired real time data. The resident programs use Persistent Data. It is unique to the mission or the systems on-board and is maintained in non-volatile memory when the system is powered down. (Example: Round Zero, and Total Engine hours) Non-Persistent Data is time sensitive, it is stored in volatile memory and is typically re-sampled at initialization. (Example: wind speed, vehicle speed, and target-lead angle.)

Step 5: Dispatched Probes validate the performance of the surrogate. The Health Controller will bring the surrogate on-line and retire the incumbent at time consistent with the operational state and usage of the element.

The DART Probe Controller will employ the following tool: the Probe as an agent of the DART Controller, reporting the health of the weapon system elements. Off-vehicle probes will be also launched to assess the health of companion vehicles within the Operations Unit. Figure 5 illustrates the DART controller system health check process. Selected software components of soldier machine interface in a crew station will be modeled using DART architecture modeling techniques. The hardware environment will be modeled so that DART analysis tools can select compatible hosts from candidate processors. Missions will also be modeled so that DART tools can make intelligent choices considering the task criticality. DART models will automatically insert software probes into the crew station to monitor the system behavior.

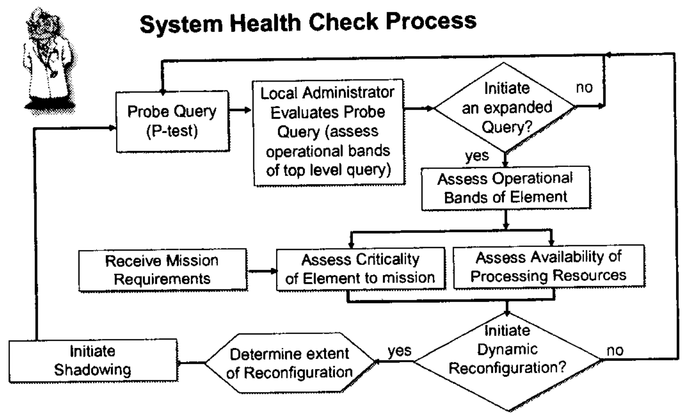

Figure 5: DART controller system health check process.

Gauges will determine if the system is operating within acceptable performance bands by monitoring data provided by the probes. DART will detect faults and select the optimal crew station configuration to maintain essential functionality in response to current battlefield conditions.

The experiment will inject artificial faults into the system according to known and anticipated patterns common to the weapon platform. The induced faults will reach a magnitude that will ultimately force DART to replicate a new processing element, bringing it on line as a replacement to the failed element. The architectural models of the system software are part of the build standards that provide the software framework for processing elements expected to reside on a future military vehicles. The build standards will document structure of the reconfigured software and hosting processors. Failures will be inserted into the Systems Integration Lab (SIL) to exercise and validate each of the build standard requirements.

DART will construct correct configurations of software to load onto a vehicle for combinations of weapons systems, sensors, and missions. It will collect usage and runtime error data that can be used to improve the software development and testing processes. DART models will automatically insert software probes into the crew station to monitor the system behavior. Gauges will determine if the system is operating within acceptable performance bands by monitoring data provided by the probes. DART will detect faults and select the optimal crew station configuration to maintain essential functionality in response to current battlefield conditions. DART-collected usage information and runtime error patterns will be fed back into Next Generation SUT models to improve the modeling fidelity and software testing process. Success of this aspect of DART will be measured by the reduction in time for the

SUT models to identify, isolate and repair errors. DART architecture descriptions will be used to improve SUT usage modeling techniques and processes. The DART process can be summarized as follows, see Figure 6.

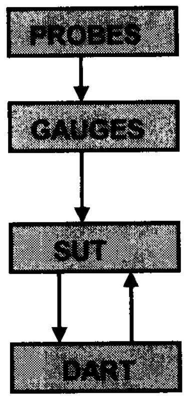

Figure 6: The DART Process.

DISCUSSIONS AND CONCLUSIONS

Software reliability and prognostics modeling have been described in this work, with particular reference to military vehicular systems. The conceptual architecture and algorithm to implement the methodologies have been shown. As indicated earlier in the introduction section, there has been some work done in the area of software reliability. The formidable tasks in those areas relate to the establishment of operational profiles. Although researchers have indicated testing methodologies using various concepts, those have not been implemented in reality. In addition, it seems that not much work has been done in the area of embedded systems and their reconfiguration when the reliability of the software is detected to be low. This work relates to both testing and reconfiguration in a dynamic manner and is important for safety critical systems. This work provides a vision of the future possible architecture and is believed to be an addition to the current state of the art.

Based on the work, the authors feel that it is possible to implement a reconfiguration methodology to an embedded system with a significant redundancy to the system without additional expenditure, if the probes are included as part of the initial design.

The authors will try to show some specific examples of implementations in their future publications.

REFERENCES

1. J. R. Horgan, A.P. Mathur, A. Pasquini, and V.J. Rego, "Perils of Software Reliability Modeling, Purdue University Research Report, 1995.
2. J. D. Musa, "Operational Profiles in Reliability Engineering", IEEE Software, March 1993.
3. E. Nelson, "Estimating Software Reliability from Test Data", Microelectronics and Reliability, Vol. 17, 1978.
4. K. W. Miller, L.J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B.W. Murrill, and J.M. Voas, "Estimating the Probability of Failure When Testing Reveals No Failures", IEEE Trans. On Software Engineering, Vol. SE-18, No 1, 1992.
5. M. S. Saboe, P. Gilbert, A. Kouchakdjian, "Applying Statistical Usage Testing (SUT) on a High-Complexity Application", *Proceedings of the Workshop on Statistical Methods in Software Engineering for Defense Systems*, National Academy of Sciences, Washington DC, July 2001.
6. E. Bankowski, C. Miles, M. Saboe, "Health Monitoring and Diagnostics of Ground Combat Vehicles", Proceedings of SPIE, Vol. 5049, pp. 138-145, 2003.
7. C. Miles, E. Bankowski, "Embedded Diagnostics in Combat Systems", Proceedings of SPIE, Vol. 5391, pp 158-165, 2004.

CONTACT

Dr. Elena N. Bankowski
TARDEC, AMSRD-TAR-R, MS-263
6501 E. 11 Mile Road
Warren, MI 48397-5000, USA
E-Mail: elena.n.bankowski@us.army.mil

Dr. M. Abul Masrur
US Army TACOM, AMSRD-TAR-R, MS-264
6501 E. 11 Mile Road
Warren, MI 48397-5000, USA
E-Mail: masrura@tacom.army.mil

Mr. Christopher Miles
US Army TACOM, AMSRD-TAR-R, MS-264
6501 E. 11 Mile Road
Warren, MI 48397-5000, USA
E-Mail: milesc@tacom.army.mil